*// Last update: Finished chapter 12.*

**CHAPTER 10: ARRAYS**

Arrays are sequences of objects of the ***same type,*** *for example we can make an array that can store integers:*

*int main()  
{  
 int arr[10];  
}*

The above array can store 10 variables of integer type, and elements are adjacent to each other (or their addresses are adjacent to each other.)

To initialize an array, we can use the **initialization list {}**

*int main()  
{  
int arr[4] = {32, -12, 0, 4};  
}*

Elements are put in the curly brackets **{ },** and each elements are separated by the comma “ **,** ”.

The order of the elements is from the left to the right: 32 is the first element, -12 is the second element, 0 is the third element and 4 is the last element.

However, it should be noted that array indexing **starts from 0, not 1.** So when you want to use the first element of **arr**, type **arr[0].**

* If the length of the array is **N**, then the index starts **from 0 to N-1.**

We can insert elements to the array from user input by using loop such as **for loop**

*int main(){*

*int arr[10]; // Create an array of integers that has the length of 10*

*for(int i=0; i<10; ++i)*

*cin >> arr[i];*

*}*

**CHAPTER 11: POINTERS**

Every objects are store in memory, so each objects has its own **value** and **address**. When talking about how to store the **address** of that object, we use a special variable called **pointers.** So a pointer is a type that can hold the address of a particular object.

To create a pointer, we use the “\*” mark before the pointer’s name.

*int main()  
{  
int \*ptr;  
}*

Above is a pointer that can hold an **integer’s address**. Conversely, we can make a pointer that can hold **char’s address.**

*int main()  
{  
char \*ptr;  
}*

To make a pointer points to a specific variable, we use the “ **&** ” operator to get the variable’s address.

int main()  
*{  
int x = 123;  
int\* p = &x;  
}*

We can also initialize our pointer to let it points to **nowhere**, or **nullptr**.

*int main()  
{  
char\* p = nullptr;  
}*

p is now called a ***null pointer***.

To access the value of an object which is being pointed to by a pointer, we can use the “ **\*** ” operator.

*#include <iostream>*

*using namespace std;*

*int main(){*

*int a = 5;*

*int \*ptr = &a;*

*cout << “The value of a is:” << \*ptr; // Prints out 5*

*}*

We can also change the value of the object through pointer.

#include <iostream>

using namespace std;  
int main()  
{  
int x = 123;  
int\* p = &x;  
\*p = 456; // change the value of pointed-to object  
cout << "The value of x is: " << x; // Prints out 456  
}

**CHAPTER 12: REFERENCES**

A reference type is an alias to an existing object in memory, and reference must be initialized. For example:

*int main()  
{  
 int x = 65;  
 int& y = x;  
}*

Now we have two different names that refer to the same integer. If we assign or change value one of them, the value of both will be changed. In other words, they are using two different names for the same value.

*int main()  
{  
 int x = 123;  
 int& y = x;  
 x = 456;  
 // both x and y now hold the value of 456  
 y = 789;  
 // both x and y now hold the value of 789  
}*

Note: The “ **\*** ” operator is used when creating a pointer:

int \*ptr;

**But**, it is also used to refer to the value of the object being pointed by a pointer.

\*ptr = 350;

The “ **&** ” operator is used to get the address of an object;

ptr = &myVar;

**But**, it is also used as initializing a reference.

int &y = x;

* It is important to look at the **context** to see what role does these operators play.

**CHAPTER 13: INTRODUCTION TO STRINGS.**

C++ library offers a compound type called string or rather std::string as it is part of the std namespace. We use it for storing and manipulating string of texts.

**Defining a string:**

To use the std::string type, we need to include the <string> header in our program.

*#include <iostream>  
#include <string>  
int main()  
{  
std::string s = "Hello World.";  
std::cout << s;  
}*

**Concatenating strings:**

We can add a string literal to our string using the compound operator “**+=**”

*#include <iostream>  
#include <string>  
int main()  
{  
std::string s = "Hello ";  
s += "World.";  
std::cout << s;  
}*

We can also connect 2 (or more) strings together with “**+**”

*#include <iostream>  
#include <string>  
int main()  
{  
std::string s1 = "Hello ";  
std::string s2 = "World.";  
std::string s3 = s1 + s2;  
std::cout << s3; // Prints out Hello World.  
}*

**Accessing Characters:**

Individual characters of a string can be accessed through a **subscript operator []** or via a  
member function ***.at(index)*.** The index starts at 0. Example:*#include <iostream>  
#include <string>  
int main()  
{  
std::string s = "Hello World.";*

|  |  |
| --- | --- |
| *char c1 = s[0];  char c2 = s.at(0);  char c3 = s[6];  char c4 = s.at(6);* | *// 'H' // 'H'; // 'W' // 'W';* |

*std::cout << "First character: " << c1 << ", sixth character: " << c3;  
}*

**Comparing Strings:**

A string can be compared to string literals and other strings using the equality “**==**” operator. Comparing a string to a string literal: *#include <iostream>  
#include <string>  
int main()  
{  
 std::string s1 = "Hello";  
 if (s1 == "Hello")  
 {  
 std::cout << "The string is equal to \"Hello\"";  
 }  
}*

*Comparing a string to another string is done using the equality operator “****==****”  
#include <iostream>  
#include <string>  
int main()  
{  
 std::string s1 = "Hello";  
 std::string s2 = "World.";  
 if (s1 == s2)  
 {  
 std::cout << "The strings are equal.";  
 }  
 else  
 {  
 std::cout << "The strings are not equal.";  
 }  
}*

***String input:***

Preferred way of accepting a string from the standard input is via the *std::getline*  
function which takes std::cin and our string as parameters:*#include <iostream>  
#include <string>  
int main()  
{  
std::string s;  
std::cout << "Please enter a string: ";  
std::getline(std::cin, s);  
std::cout << "You entered: " << s;  
}*

We use the *std::getline* because our string can contain white spaces. And if we  
used the *std::cin* function alone, it would accept only a part of the string.

**A pointer to a string:**

A string has a member function .c\_str() which returns a pointer to its first element.

#include <iostream>  
#include <string>  
int main()  
{  
 std::string s = "Hello World.";  
 std::cout << s.c\_str();  
}

**Substrings:**

To create a substring from a string, we use the *.substr()* member function. The signature of the function is: *.substring(starting\_position, length) .*Example:

#include <iostream>  
#include <string>  
int main()  
{  
 std::string s = "Hello World.";  
 std::string mysubstring = s.substr(6, 5);  
 std::cout << "The substring value is: " << mysubstring; // Prints out World  
}

**Finding a substring:**

To find a substring in a string, we use the *.find()* member function. If the substring is found, the function returns the position of the first found substring. If the substring is not found, the function returns a value that is *std::string::npos*.

To find a substring “Hello” inside the “This is a Hello World string” string, we write:  
#include <iostream>  
#include <string>  
int main()  
{  
 std::string s = "This is a Hello World string.";  
 std::string stringtofind = "Hello";  
 std::string::size\_type found = s.find(stringtofind);  
 if (found != std::string::npos)  
 {  
 std::cout << "Substring found at position: " << found;  
 }  
 else  
 {  
 std::cout << "The substring is not found.";  
 }  
}

**CHAPTER 14: AUTOMATIC TYPE DEDUCTION**

We can automatically deduce the type of an object using the **auto** specifier. The **auto**specifier deduces the type of an object based on the object’s **initializer type.**Example:

|  |  |
| --- | --- |
| auto c = 'a'; | // char type |
| This example deduces c to be of type char as the initializer 'a' is of type char. Similarly, we can have: |  |
| auto x = 123; | // int type |

Here, the compiler deduces the x to be of type int because an integer literal 123 is of type int.  
The type can also be deduced based on the type of expression:  
auto d = 123.456 / 789.10; // double

This example deduces d to be of type double as the type of the entire 123.456 / 789.10 expression is double.  
We can use auto as part of the reference type:  
int main()  
{  
int x = 123;  
auto& y = x; // y is of int& type  
}

or as part of the constant type:  
int main()  
{  
const auto x = 123; // x is of const int type  
}

We use the *auto* specifier when the type (name) is hard to deduce manually or cumbersome to type due to the length.